Quatrième partie

Probabilités, combinatoire et statistiques

12

Dénombrement et combinatoire

Ce chapitre aborde principalement le traitement avec Sage des problèmes combinatoires suivants : le dénombrement (combien y-a-t'il d'éléments dans S ?), l'énumération (calculer tous les éléments de S, ou itérer parmi eux), le tirage aléatoire (choisir au hasard un élément de S selon une loi, mettons uniforme). Ces questions interviennent naturellement dans les calculs de probabilités (quelle est la probabilité au poker d'obtenir une suite ou un carré d'as ?), en physique statistique, mais aussi en calcul formel (nombre d'éléments dans un corps fini), ou l'analyse d'algorithmes. La combinatoire couvre un domaine beaucoup plus vaste (graphes, ordres partiels, théorie des représentations, ...) pour lesquels nous nous contenterons de donner quelques pointeurs vers les possibilités offertes par Sage.

Une caractéristique de la combinatoire effective est la profusion de types d'objets et d'ensembles que l'on veut manipuler. Il serait impossible de les décrire et a fortiori de les implanter tous. Ce chapitre illustre donc la méthodologie sous-jacente : fournir des briques de bases pour décrire les ensembles combinatoires usuels 12.2, des outils pour les combiner et construire de nouveaux ensembles 12.3, et des algorithmes génériques pour traiter uniformément de grandes classes de problèmes 12.4.

C'est un domaine où Sage a des fonctionnalités bien plus étendues que la plupart des systèmes de calcul formel et est en pleine expansion ; en revanche il reste encore très jeune avec de multiples limitations arbitraires et incohérences.

12.1 Premiers exemples

12.1.1 Jeu de Poker et probabilités

Nous commençons par résoudre un problème classique : dénombrer certaines combinaisons de cartes dans un jeu de Poker, pour en déduire leur probabilité.

Une carte de Poker est caractérisée par une couleur (cœur, carreau, pique ou trèfle) et une valeur (2,3, . . .,9, valet, dame, roi, ou as). Le jeu de Poker est constitué de toutes les cartes possibles ; il s'agit donc du produit cartésien de l'ensemble des couleurs et de l'ensemble des valeurs :

$$\text{Cartes} = \text{Symboles} \times \text{Valeurs} = \{(c, v) \quad \| c \in \text{Symboles} \text{ et } v \in \text{Valeurs}\}$$

Construisons ces ensembles dans Sage :

```
sage: Symboles = Set(["Coeur", "Carreau", "Pique", "Trefle"])
```

```
sage: Valeurs = Set([2,3,4,5,6,7,8,9,10, "Valet", "Dame", "Roi", "As"])
```

```
sage: Cartes = CartesianProduct(Symboles, Valeurs)
```

Il y a 4 couleurs et 13 valeurs possibles donc $4 \times 13 = 52$ cartes dans le jeu de Poker :

```
sage: Symboles.cardinality()
```

$$4$$

```
sage: Valeurs.cardinality()
```

$$13$$

```
sage: Cartes.cardinality()
```

$$52$$

Tirons une carte au hasard :

```
sage: Cartes.random_element()
```

$$[\texttt{Pique}, 9]$$

Une petite digression technique. Les éléments du produit cartésien sont actuellement renvoyés sous forme de listes :

```
sage: type(Cartes.random_element())
```

```
<type 'list'>
```

Une liste Python n'étant pas immuable, on ne peut pas la mettre dans un ensemble, ce qui nous poserait problème par la suite (voir § 3.2.7). Nous redéfinissons donc notre produit cartésien pour que les éléments soient représentés par des tuples :

```
sage: Cartes = CartesianProduct(Symboles, Valeurs).map(tuple)
```

```
sage: Cartes.random_element()
```

$$(\texttt{Trefle}, 8)$$

On peut maintenant construire un ensemble de cartes :

```
sage: Cartes = CartesianProduct(Symboles, Valeurs).map(tuple)
```

```
sage: Set([Cartes.random_element(), Cartes.random_element()])
```

$$\{(\texttt{Pique}, 2), (\texttt{Coeur}, 4)\}$$

Revenons à notre propos. Au début d'une partie de Poker, chaque joueur pioche cinq cartes, qui forment une *main*. Toutes les cartes sont distinctes et l'ordre n'est pas relevant ; une main est donc un sous ensemble de taille 5 de l'ensemble des cartes. Pour tirer une main au hasard, on commence par construire l'ensemble de toutes les mains possibles puis on en demande un élément aléatoire :

```
sage: Mains = Subsets(Cartes, 5)
```

```
sage: Mains.random_element()
```

$$\{(\texttt{Pique}, 2), (\texttt{Carreau}, \texttt{Valet}), (\texttt{Carreau}, 4), (\texttt{Trefle}, \texttt{Valet}), (\texttt{Trefle}, 6)\}$$

Le nombre total de mains est donné par le nombre de sous-ensembles de taille 5 d'un ensemble de taille 52, c'est-à-dire le coefficient binomial $\binom{52}{5}$:

```
sage: binomial(52,5)
```

$$2598960$$

On peut aussi ne pas se préoccuper de la méthode de calcul, et simplement demander sa taille à l'ensemble des mains :

```
sage: Mains.cardinality()
```

$$2598960$$

La force d'une main de Poker dépend de la combinaison de ses cartes. Une de ces combinaisons est la *couleur* ; il s'agit d'une main dont toutes les cartes ont le même symbole (en principe il faut exclure les quintes flush ; ce sera l'objet d'un exercice ci-dessous). Une telle main est donc caractérisée par le choix d'un symbole parmi les quatre possibles et le choix de cinq valeurs parmi les treize possibles. Construisons l'ensemble de toutes les couleurs, pour en calculer le nombre :

```
sage: Couleurs = CartesianProduct(Symboles, Subsets(Valeurs, 5))
```

```
sage: Couleurs.cardinality()
```

$$5148$$

La probabilité d'obtenir une couleur en tirant une main au hasard est donc de :

```
sage: Couleurs.cardinality()  / Mains.cardinality()
```

$$\frac{33}{16660}$$

soit d'environ deux sur mille :

```
sage: 1000.0 * Couleurs.cardinality()  / Mains.cardinality()
```

1.98079231692677

Faisons une petite simulation numérique. La fonction suivante teste si une main donnée est une couleur :

```
sage: def est_couleur(main):
....:     return len(set(couleur for (couleur, valeur) in main)) == 1
```

Nous tirons maintenant 10000 mains au hasard, et comptons le nombre de couleurs obtenues (cela prend environ 10s) :

```
sage: n = 10000

sage: ncouleurs = 0

sage: for i in range(n):
....:     main = Mains.random_element()
....:     if est_couleur(main):
....:         ncouleurs += 1

sage: print n, ncouleurs
```

Exercice 36. Une main contenant quatre cartes de la même valeur est appelée un *carré*. Construire l'ensemble des carrés (indication : utiliser `Arrangements` pour tirer au hasard un couple de valeurs distinctes puis choisir une couleur pour la seconde valeur). Calculer le nombre de carrés, en donner la liste, puis déterminer la probabilité d'obtenir un carré en tirant une main au hasard.

Exercice 37. Une main dont les cartes ont toutes le même symbole et dont les valeurs se suivent est en fait appelée une *quinte flush* et non une *couleur*. Compter le nombre de quintes flush, puis en déduire la probabilité correct d'obtenir une couleur en tirant une main au hasard.

Exercice 38. Calculer la probabilité de chacune des combinaisons de cartes au Poker (voir http://fr.wikipedia.org/wiki/Main_au_poker) et comparer avec le résultats de simulations.

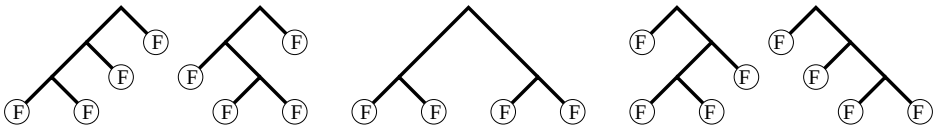

FIGURE 12.1: Les cinq arbres binaires complets à quatre feuilles

12.1.2 Dénombrement des arbres binaires complets par séries génératrices

Dans cette section, nous traitons l'exemple des arbres binaires complets, et illustrons sur cet exemple plusieurs techniques de dénombrement où le calcul formel intervient naturellement. Ces techniques sont en fait générales, s'appliquant à chaque fois que les objets combinatoires considérés admettent une définition récursive (grammaire) (voir § **??** pour un traitement automatisé). L'objectif n'est pas de présenter ces méthodes formellement ; aussi les calculs seront rigoureux mais la plupart des justifications seront passées sous silence.

Un *arbre binaire complet* est soit une feuille, soit un noeud sur lequel on a greffé deux arbres binaires complets (voir figure 12.1).

Exercice 39. Chercher à la main tous les arbres binaires complets à $n = 1, 2, 3, 4, 5$ feuilles (voir l'exercice 47 pour les chercher avec Sage).

Notre objectif est de compter le nombre c_n d'arbres à n feuilles ; d'après l'exercice précédent, les premiers termes sont donnés par $c_1, \ldots, c_5 = 1, 1, 2, 5, 14$. Le simple fait d'avoir ces quelques nombres est déjà précieux. En effet, ils vont nous permettre une recherche dans une mine d'or : l'*encyclopédie en ligne des suites de nombres entiers* http://www.research.att.com/~njas/sequences/ appelée communément le Sloane, du nom de son auteur principal, et qui contient plus de 170000 suites d'entiers :

```
sage: sloane_find([1,1,2,5,14])
Searching Sloane's online database...
[[108, 'Catalan numbers: C(n) = binomial(2n,n)/(n+1) ...
```

Le résultat suggère que les arbres binaires complets sont comptés par l'une des plus fameuses suites, les nombres de Catalan. En fouillant, dans les références, on trouverait que c'est effectivement le cas. En résumé, ces quelques nombres ci-dessus forment une empreinte digitale de nos objets, qui nous on permis de retrouver en quelques secondes un résultat dans l'abondante littérature sur le sujet. L'objectif de la suite est de retrouver ce résultat à la main.

Soit C_n l'ensemble des arbres binaires complets à n feuilles, de sorte que $c_n = |C_n|$; par convention, on définira $C_0 = \emptyset$. L'ensemble de tous les arbres est alors la réunion disjointe des C_n :

$$C = \biguplus_{n \in \mathbb{N}} C_n \,.$$

Du fait d'avoir nommé l'ensemble C de tous les arbres, on peut traduire la definition récursive des arbres en une équation ensembliste :

$$C \quad \approx \quad \{F\} \quad \uplus \quad C \times C$$

En mots : un arbre t (donc dans C) est soit une feuille (donc dans $\{F\}$) soit un noeud sur lequel on a greffé deux arbres t_1 et t_2 et que l'on peut donc identifier avec le couple (t_1, t_2) (donc dans le produit cartésien $C \times C$).

L'idée fondatrice de la combinatoire algébrique, introduite par Euler pour traiter un problème similaire [**?**], est de manipuler simultanément tous les coefficients c_n en les encodant sous la forme d'une série formelle, dite *série génératrice* des c_n :

$$C(z) = \sum_{n \in \mathbb{N}} c_n z^n$$

où z est une indéterminée formelle (on n'a donc pas besoin de se préoccuper de questions de convergence). La beauté de cette idée est que les opérations ensemblistes ($A \uplus B$, $A \times B$) se traduisent naturellement en opérations algébriques sur les séries ($A(z) + B(z)$, $A(z)\dot{B}(z)$), de sorte que l'équation ensembliste vérifiée par C se traduit en une équation algébrique sur $C(z)$:

$$C(z) = 1 + C(z)\dot{C}(z)$$

Résolvons cette équation avec Sage. Pour cela, on introduit deux variables C et z, et on pose le système :

```
sage: var("C,z");

sage: sys = [C == z + C*C]

sage: sol = solve(sys, C, solution_dict=True)

sage: sol
```

$$\left[\left\{C : -\frac{1}{2}\sqrt{-4\,z+1} + \frac{1}{2}\right\}, \left\{C : \frac{1}{2}\sqrt{-4\,z+1} + \frac{1}{2}\right\}\right]$$

On a alors deux solutions :

```
sage: s0 = sol[0][C]

sage: s1 = sol[1][C]
```

Dont les développements de Taylor commencent par :

```
sage: taylor(s0, z, 0, 5)
```

$$14\,z^5 + 5\,z^4 + 2\,z^3 + z^2 + z$$

```
sage: taylor(s1, z, 0, 5)
```

$$-14\,z^5 - 5\,z^4 - 2\,z^3 - z^2 - z + 1$$

La deuxième solution est clairement aberrante ; par contre, on retrouve les coefficients prévus sur la première. Posons donc :

```
sage: C = s0
```

On peut maintenant calculer les termes suivants :

```
sage: taylor(C, z, 0, 10)
```

$$4862\,z^{10} + 1430\,z^9 + 429\,z^8 + 132\,z^7 + 42\,z^6 + 14\,z^5 + 5\,z^4 + 2\,z^3 + z^2 + z$$

ou calculer quasi instantanément le 99-ème coefficient :

```
sage: taylor(C, z, 0, 100).coeff(z,99)
```

$$5774335806960135778218770060804285633402073162475661100 0$$

Le n-ième coefficient du développement de Taylor de `C(z)` étant donné par $\frac{1}{n!}C(z)^{(n)}(0)$, regardons les dérivées successives $C(z)^{(n)}$ de $C(z)$:

```
sage: derivative(C, z, 0)
```

$$-\frac{1}{2}\sqrt{-4\,z+1} + \frac{1}{2}$$

```
sage: derivative(C, z, 1)
```

$$\frac{1}{\sqrt{-4\,z+1}}$$

```
sage: derivative(C, z, 2)
```

$$\frac{2}{(-4\,z+1)^{\left(\frac{3}{2}\right)}}$$

```
sage: derivative(C, z, 3)
```

$$\frac{12}{(-4\,z+1)^{\left(\frac{5}{2}\right)}}$$

```
sage: derivative(C, z, 4)
```

$$\frac{120}{(-4\,z+1)^{\left(\frac{7}{2}\right)}}$$

Cela suggère l'existence d'une forme close simple que l'on recherche maintenant. La petite fonction suivante renvoie $d_n = n!c_n$:

```
sage: def d(n): return derivative(s0, n).subs(z=0)
```

En en prenant les quotients successifs :

```
sage: [ (d(n+1) / d(n)) for n in range(1,20) ]
```

$$[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, 66, 70, 74]$$

on constate que d_n satisfait la relation de récurrence $d_{n+1} = (4n-2)d_n$, d'où l'on déduit que c_n satisfait la relation de récurrence $c_{n+1} = \frac{(4n-2)}{n} c_n$. En simplifiant, on obtient alors que c_n est le $(n-1)$-ième nombre de Catalan :

$$c_n = \text{Catalan}(n-1) = \frac{1}{n}\binom{2(n-1)}{n-1}.$$

Vérifions cela :

```
sage: var('n')
```

$$n$$

```
sage: c = 1/n*binomial(2*(n-1),n-1)

sage: [c.subs(n=k) for k in range(1, 11)]
```

$$[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]$$

```
sage: [catalan_number(k-1) for k in range(1, 11)]
```

$$[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]$$

On peut maintenant calculer les coefficients beaucoup plus loin ; ici on calcule $c_{1000000}$:

```
sage: %time x = c(100000);
CPU times: user 2.34 s, sys: 0.00 s, total: 2.34 s
Wall time: 2.34 s
```

qui as plus de 60000 chiffres :

```
sage: len(str(x))
60198
```

Les méthode que nous avons utilisées se généralisent à tous les objets définis récursivement : le système d'équation ensembliste se traduit en un système d'équation sur la série génératrice ; celui-ci permet de calculer récursivement ses coefficients. Lorsque les équations ensemblistes sont suffisamment simple (par exemple ne font intervenir que des produits cartésiens et unions disjointes), l'équation sur $C(z)$ est algébrique. Elle admet rarement une solution en forme close, mais en calculant les dérivées successives, on peut en déduire une équation différentielle *linéaire* sur $C(z)$, qui se traduit en une équation de récurrence de longueur fixe sur les coefficients c_n (la série est alors dite *holonomique*). Au final, après le précalcul de cette équation de récurrence, le calcul des coefficients devient très rapide.

12.2 Ensembles énumérés usuels

12.2.1 Premier exemple : les sous-ensembles d'un ensemble

Fixons un ensemble E de taille n et considérons les sous-ensembles de E de taille k de E. On sait que ces sous-ensembles sont comptés par les coefficients binomiaux $\binom{n}{k}$. On peut donc calculer le nombre de sous-ensembles de taille $k = 2$ de $E = \{1, 2, 3, 4\}$ avec la fonction `binomial` :

```
sage: binomial(4, 2)
```

$$6$$

Alternativement, on peut *construire* l'ensemble $\mathcal{P}_2(E)$ de tous les sous-ensembles de taille 2 de E, puis lui demander sa cardinalité :

```
sage: S = Subsets([1,2,3,4], 2)
```

```
sage: S.cardinality()
```

$$6$$

Une fois `S` construit, on peut aussi obtenir la liste de ses éléments :

```
sage: S.list()
```

$$[\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}]$$

ou tirer un élément au hasard :

```
sage: S.random_element() # random
```

$$\{1,3\}$$

Plus précisément, l'objet `S` modélise l'ensemble $\mathcal{P}_2(E)$, muni d'une énumération fixée (donnée ici par l'ordre lexicographique). On peut donc demander quel est son 5-ème élément :

```
sage: S.unrank(4)
```

$$\{2,4\}$$

À titre de raccourci, on peut utiliser ici la notation :

```
sage: S[4]
```

$$\{2,4\}$$

mais cela est à utiliser avec prudence car certains ensembles sont munis d'une indexation naturelle autre que par $(0, \ldots,)$.

Réciproquement, on peut calculer le rang d'un objet dans cet énumération :

```
sage: S.rank([2, 4])
```

4

(comme pour les listes Python, le premier élément est de rang 0).

À noter que `S` n'*est pas* la liste de ses éléments. On peut par exemple modéliser l'ensemble $\mathcal{P}(\mathcal{P}(\mathcal{P}(E)))$ et calculer sa cardinalité ($2^{2^{2^4}}$) :

```
sage: E = Set([1,2,3,4])
sage: S = Subsets(Subsets(Subsets(E)))
sage: S.cardinality()
200352993040684646497907235156025575044782547556975141926501697371089405 95
...736L
```

soit environ 210^{19728} :

```
sage: len(str(S.cardinality()))
19729
```

ou demander son 237102123-ième élément :

```
sage: S.unrank(237102123)
{{{2, 4}, {1, 2}, {1, 4}, {}, {2, 3, 4}, {3, 4}, {1, 3, 4}, {1}, {4}},
 {{2, 4}, {3, 4}, {1, 2, 3, 4}, {1, 2, 3}, {}, {2, 3, 4}}}
```

Il serait physiquement impossible de construire explicitement tous ses éléments car il y en aurait bien plus que de particules dans l'univers (estimé à 10^{82}).

Remarque : il serait naturel avec Python d'utiliser `len(S)` pour demander la cardinalité de `S`. Cela n'est pas possible car Python impose que le résultat de `len` soit un entier de type `int`, dans lequel le nombre ci-dessus ne rentrerait pas sans débordement.

```
sage: len(S)
...
AttributeError: __len__ has been removed; use .cardinality() instead
```

12.2.2 Partitions d'entiers

Nous considérons maintenant un autre problème classique : étant donné un entier positif n, de combien de façons peut-on l'écrire sous la forme d'une somme $n = i_1 + i_2 + \cdots + i_k$, où $i_1, \ldots, i_k$ sont des entiers strictement positifs. Il y a deux cas à distinguer :

- l'ordre des éléments dans la somme est relevant, auquel cas $(i_1, \ldots, i_k)$ est une *composition* de n ;
- l'ordre des éléments dans la somme n'est pas relevant et $(i_1, \ldots, i_k)$ est une *partition* de n.

Regardons pour commencer les partitions de $n = 5$; comme précédemment, on commence par construire l'ensemble de ces partitions :

```
sage: P5 = Partitions(5)

sage: P5

                    Partitions of the integer 5
```

puis on lui demande sa cardinalité :

```
sage: P5.cardinality()

                                    7
```

Regardons ces 7 partitions ; l'ordre n'étant pas relevant, les entrées sont triées, par convention, par ordre décroissant.

```
sage: P5.list()

[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
```

Le calcul du nombre de partitions utilise la formule de Rademacher, qui plus est implantée en `C` et fortement optimisée, ce qui lui confère une grande rapidité :

```
sage: Partitions(100000).cardinality()

2749351056977569651267751632098635268817342931598005475820312598430214732811496417305505074166073662159
```

Les partitions d'entiers sont des objets combinatoires naturellement munies de multiples opérations. Elles sont donc renvoyées sous la forme d'objets plus riches que de simples listes :

```
sage: P7 = Partitions(7)

sage: p = P7.unrank(5)

sage: p

                               [4, 2, 1]

sage: type(p)

        <class 'sage.combinat.partition.Partition_class'>
```

On peut par exemple les représenter graphiquement par un diagramme de Ferrers :

```
sage: print p.ferrers_diagram()
```

Nous laissons l'utilisateur explorer par introspection les possibilités offertes.
À noter que l'on peut aussi construire une partition directement avec :

```
sage: Partition([4,2,1])

                               [4, 2, 1]
```

ou bien :

```
sage: P7([4,2,1])
```

```
[4, 2, 1]
```

Le cas des compositions d'entiers se traite de la même façon :

```
sage: C5 = Compositions(5)
```

```
sage: C5
```

```
Compositions of 5
```

```
sage: C5.cardinality()
```

16

```
sage: C5.list()
```

```
[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3], [1, 2, 1, 1], [1, 2, 2], [1, 3, 1], [1,
```

Le 16 ci-dessus ne paraît pas anodin et suggère l'existence d'une éventuelle formule. Regardons donc le nombre de compositions de n pour n variant de 0 à 9 :

```
sage: [ Compositions(n).cardinality() for n in range(10) ]
```

$$[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]$$

De même, si l'on compte le nombre de compositions de 5 par longueur, on retrouve une ligne du triangle de Pascal :

```
sage: sum(x^len(c) for c in C5)
```

$$x^5 + 4x^4 + x^3 + 4x^2 + x$$

L'exemple ci-dessus utilise une fonctionnalité que l'on n'avait pas encore croisé : `C5` étant itérable, on peut l'utiliser comme une liste dans une boucle `for` ou une compréhension **??**.

Exercice 40. Démontrer la formule suggérée par les exemples ci-dessus pour le nombre de compositions de n et le nombre de compositions de n de longueur k et chercher par introspection si Sage utilise ces formules pour le calcul de cardinalité.

12.2.3 Quelques autres ensembles finis énumérés

Au final, le principe est le même pour tous les ensembles finis sur lesquels on veut faire de la combinatoire avec Sage ; on commence par construire un object qui modélise cet ensemble puis on utilise les méthodes idoines qui suivent une interface uniforme (où en tous cas cela devrait être le cas ; il reste de nombreux coins à nettoyer). Nous donnons maintenant quelques autres exemples typiques.

Les intervalles d'entiers :

```
sage: C = IntegerRange(3, 13, 2)
```

```
sage: C
```

```
3, 5 .. 11
```

```
sage: C.cardinality()
```

5

```
sage: C.list()
```

$[3, 5, 7, 9, 11]$

Les permutations :

```
sage: C = Permutations(4)
sage: C
```

```
Standard permutations of 4
```

```
sage: C.cardinality()
```

24

```
sage: C.list()
```

[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4

Les partitions ensemblistes :

```
sage: C = SetPartitions([1,2,3])
sage: C
```

```
Set partitions of [1, 2, 3]
```

```
sage: C.cardinality()
```

5

```
sage: C.list()
```

$[\{\{1,2,3\}\}, \{\{2,3\}, \{1\}\}, \{\{1,3\}, \{2\}\}, \{\{1,2\}, \{3\}\}, \{\{2\}, \{3\}, \{1\}\}]$

Les ordres partiels sur 5 sommets, à un isomorphisme près :

```
sage: C = Posets(5)
sage: C.cardinality()
```

63

```
sage: C.random_element().plot()
```

À noter que l'on peut lister les 34 graphes simples à un isomorphisme près mais pas encore construire l'ensemble de ces graphes :

```
sage: len(list(graphs(5)))
```

$$34$$

Voici tous les graphes à 5 sommets et moins de 4 arêtes :

```
sage: show(graphs(5, lambda G: G.size() <= 4))
```

Ce que l'on a vu s'applique aussi en principe aux structures algébriques finies comme le groupe dihédral :

```
sage: G = DihedralGroup(4)
```

```
sage: G
```

$$\langle (1,2,3,4), (1,4)(2,3) \rangle$$

```
sage: G.cardinality()
```

$$8$$

```
sage: G.list()
```

$$[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3), (1,3)(2,4), (1,4,3,2), (1,4)(2,3)]$$

ou l'algèbre des matrices 2×2 sur le corps fini $\mathbb{Z}/2\mathbb{Z}$:

```
sage: C = MatrixSpace(GF(2), 2)
```

```
sage: C.list()
```

$$\left[\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

Ceci ne fonctionne pas encore :

```
sage: C.cardinality()
16
```

Exercice 41. Lister tous les monômes de degré 5 dans les polynômes en trois variables (voir `IntegerVectors`). Manipuler les partitions ensemblistes ordonnées (`OrderedSetPartitions`) et les tableaux standard (`StandardTableaux`).

Exercice 42. Lister les matrices à signe alternant de taille 3, 4 et 5 (`AlternatingSignMatrices`), et essayer de deviner leur définition.

Exercice 43. Calculer à la main le nombre de vecteurs dans $(\mathbb{Z}/2\mathbb{Z})^5$ puis le nombre de matrices dans $GL_3(\mathbb{Z}/2\mathbb{Z})$ (c'est-à-dire le nombre de matrice 3×3, à coefficients dans $\mathbb{Z}/2\mathbb{Z}$ et inversibles). Vérifier votre réponse avec Sage. Généraliser à $GL_n(\mathbb{Z}/q\mathbb{Z})$.

12.2.4 Compréhensions et itérateurs

Nous allons maintenant montrer quelques possibilités offertes par Python pour construire (et itérer sur) des ensembles avec une notation flexible et proche des mathématiques, et le profit que l'on peut en tirer en combinatoire.

Commençons par construire l'ensemble fini $\{i^2 \mid i \in \{1,3,7\}$:

```
sage: [ i^2 for i in [1, 3, 7] ]
```

$$[1, 9, 49]$$

puis le même ensemble avec i variant cette fois entre 1 et 9 :

```
sage: [ i^2 for i in range(1,10) ]
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81]$$

On appelle *compréhension* une telle construction Python. On peut rajouter un prédicat pour ne garder que les éléments du précédent avec i premier :

```
sage: [ i^2 for i in range(1,10) if is_prime(i) ]
```

$$[4, 9, 25, 49]$$

En combinant plusieurs compréhensions, on peut construire l'ensemble $\{(i,j) \mid 1 \leqslant j < i < 5\}$:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
```

$$[(2,1), (3,1), (3,2), (4,1), (4,2), (4,3), (5,1), (5,2), (5,3), (5,4)]$$

ou bien afficher le triangle de Pascal :

```
sage: [ [ binomial(n, i) for i in range(n+1) ] for n in range(10) ]
```

$$[[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1], [1,5,10,10,5,1], [1,6,15,20,15,6,1], [1,7,21,35,35,21,7,1], [1,8,28,56,70,5$$

L'exécution d'une compréhension se fait en fait en deux étapes ; tout d'abord un *itérateur* est construit, puis une liste est remplie avec les éléments renvoyés successivement par l'*érateur. Techniquement, un* itérateur *est un objet avec une méthode* `next` *qui renvoie à chaque appel une nouvelle valeur, jusqu'à épuisement. Par exemple, l'itérateur* it *suivant :*

```
sage: it = (binomial(3, i) for i in range(4))
```

renvoie successivement les coefficients binômiaux $\binom{3}{i}$ *avec* $i = 0, 1, 2, 3$ *:*

```
sage: it.next()
```

$$1$$

```
sage: it.next()
```

$$3$$

```
sage: it.next()
```

3

```
sage: it.next()
```

1

Lorsque l'itérateur est finalement épuisé, une exception est levée :

```
sage: it.next()
Traceback (most recent call last):
  ...
StopIteration
```

Dans la pratique on n'utilise que très rarement directement cette méthode next. *Un itérable est un object* Python `l` *(une liste, en ensemble, ...) sur les éléments duquel on peut itérer. Techniquement, on construit l'itérateur avec* `iter(l)`*, mais là encore on ne le fait que rarement explicitement.*

Quel est l'intérêt d'un itérateur ? Considérons l'exemple suivant :

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
```

256

À l'exécution, une liste avec 9 *éléments est construite, puis elle est passée en argument à* `sum` *pour les ajouter. Si au contraire on passe directement l'itérateur à* `sum` *(noter l'absence de crochets) :*

```
sage: sum( binomial(8, i) for i in xrange(9) )
```

256

La fonction `sum` *reçoit directement l'itérateur, et peut ainsi court-circuiter la construction de la liste intermédiaire. Lorsqu'il y a un grand nombre d'éléments, cela évite donc d'allouer une grosse quantité de mémoire pour stoker une liste qui sera immédiatement détruite*[1]

La plupart des fonctions prenant une liste d'éléments en entrée acceptent un itérateur (ou un itérable) à la place. Et pour commencer, on peut obtenir la liste (ou le tuple) des éléments d'un itérateur avec :

```
sage: list(binomial(8, i) for i in xrange(9))
```

$$[1, 8, 28, 56, 70, 56, 28, 8, 1]$$

```
sage: tuple(binomial(8, i) for i in xrange(9))
```

$$(1, 8, 28, 56, 70, 56, 28, 8, 1)$$

Prenons maintenant les fonctions `all` *and* `any` *(qui dénotent respectivement le* et *et le* ou *n-aire) :*

1. Détail technique : `xrange` renvoie un itérateur sur {0,...,8} alors que `range` en renvoie la liste. À partir de Python 3.0, `range` se comportera comme `xrange`, et on pourra oublier ce dernier.

```
sage: all([True, True, True, True])
```

True

```
sage: all([True, False, True, True])
```

False

```
sage: any([False, False, False, False])
```

False

```
sage: any([False, False, True, False])
```

True

L'exemple suivant vérifie que tous les entiers premiers entre 3 *et* 100 *exclus sont impairs :*

```
sage: all( is_odd(p) for p in range(3,100) if is_prime(p) )
```

True

Les nombres de Mersenne $m(p)$ *sont les nombres de la forme* $2^p - 1$*. Ici nous vérifions (jusqu'à* $p = 999$*, que si* $m(p)$ *est premier, alors* p *est premier aussi :*

```
sage: def mersenne(p): return 2^p -1

sage: [ is_prime(p) for p in range(1000) if is_prime(mersenne(p)) ]
```

[True, True, True, True, True, True, True, True, True, True, True, True, True]

Exercice 44. La réciproque est-elle vraie ? Essayer les deux commandes suivantes, puis expliquer la différence considérable de temps de calcul :

```
sage: all(   is_prime(mersenne(p)) for p in range(1000) if is_prime(p)  )
```

False

```
sage: all( [ is_prime(mersenne(p)) for p in range(1000) if is_prime(p)] )
```

False

On cherche maintenant à trouver le plus petit contre exemple. Pour cela on peut utiliser la fonction Sage `exists` *:*

```
sage: exists( (p for p in range(1000) if is_prime(p)), lambda p: not is_prime(mersenne(p)) )
```

(True, 11)

Alternativement, on peut construire un itérateur sur tous les contre-exemples, puis les demander un par un : :

```
sage: contre_exemples = (p for p in range(1000) if is_prime(p) and not is_prime(mersenne(p)))

sage: contre_exemples.next()
```

11

```
sage: contre_exemples.next()
```

23

Exercice 45. Que font les commandes suivantes ?

```
sage: cubes = [t**3 for t in range(-999,1000)]
```

```
sage: exists([(x,y) for x in cubes for y in cubes], lambda (x,y): x+y == 218)
```

(True, (−125, 343))

```
sage: exists(((x,y) for x in cubes for y in cubes), lambda (x,y): x+y == 218)
```

(True, (−125, 343))

Laquelle des deux dernières est-elle la plus économe en temps ? en mémoire ? De combien ?

Exercice 46. Essayer tour à tour les commandes suivantes, et expliquer leurs résultats. Attention : il sera nécessaire d'interrompre l'exécution de certaines certaines de ces commandes en cours de route.

```
sage: sum( x^len(s) for s in Subsets(8) )
```

```
sage: sum( x^p.length() for p in Permutations(3) )
```

```
sage: factor(sum( x^p.length() for p in Permutations(3) ))
```

```
sage: P = Permutations(5)
sage: all( p in P for p in P )
True
```

```
sage: for p in GL(2, 2): print p; print
```

```
sage: for p in Partitions(3): print p
```

```
sage: for p in Partitions(): print p
```

```
sage: for p in Primes(): print p
```

```
sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

```
sage: contre_exemples = (p for p in Primes() if not is_prime(mersenne(p)))
sage: for p in contre_exemples: print p
```

Opérations sur les itérateurs

Python fournit de nombreux utilitaires pour manipuler des itérateurs ; la plupart d'entre eux sont dans la bibliothèque `itertools` *que l'on peut importer avec :* `??` *On va en montrer quelques applications, en prenant comme point de départ l'ensemble des permutations de* 3 *:*

```
sage: list(Permutations(3))
```

$$[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]$$

Énumérer les éléments d'un ensemble en les numérotant :

```
sage: list(enumerate(Permutations(3)))
```

$$[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]), (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]$$

Sélectionner seulement les éléments en position 2,3 et 4 (analogue de l[1 :4]) :

```
sage: import itertools
```

```
sage: list(itertools.islice(Permutations(3), 1, 4))
```

$$[[1, 3, 2], [2, 1, 3], [2, 3, 1]]$$

Appliquer une fonction sur tous les éléments :

```
sage: list(itertools.imap(lambda z: z.cycle_type(), Permutations(3)))
```

$$[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]$$

Sélectionner les éléments vérifiant un certain prédicat :

```
sage: list(itertools.ifilter(lambda z: z.has_pattern([1,2]), Permutations(3)))
```

$$[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]$$

Implantation de nouveaux itérateurs

Il est possible de construire très facilement de nouveaux itérateurs, en utilisant le mot clef `yield` *plutôt que* `return` *dans une fonction :*

```
sage: def f(n):
....:     for i in range(n):
....:         yield i
```

À la suite du `yield`*, l'exécution n'est pas arrêtée, mais seulement suspendue, et prête à reprendre au même point. Le résultat de la fonction est alors un itérateur sur les valeurs successives renvoyées par* `yield` *:*

```
sage: g = f(4)
sage: g.next()
```

$$0$$

```
sage: g.next()
```

$$1$$

```
sage: g.next()
```

$$2$$

```
sage: g.next()
```

$$3$$

```
sage: g.next()
Traceback (most recent call last):
  ...
StopIteration
```

En utilisation courante, cela donnera :

```
sage: [ x for x in f(5) ]
```

$$[0, 1, 2, 3, 4]$$

Ce paradigme de programmation, appelé continuation *est très utile en combinatoire, surtout quand on le combine avec la récursivité. Voici comment engendrer tous les mots d'une longueur et sur un alphabet donnés :*

```
sage: def words(alphabet,l):
....:     if l == 0:
....:         yield []
....:     else:
....:         for word in words(alphabet, l-1):
....:             for l in alphabet:
....:                 yield word + [l]
```

```
sage: [ w for w in words(['a','b'], 3) ]
```

$$[[a, a, a], [a, a, b], [a, b, a], [a, b, b], [b, a, a], [b, a, b], [b, b, a], [b, b, b]]$$

On peut les compter avec :

```
sage: sum(1 for w in words(['a','b','c','d'], 10))
```

1048576

Compter les mots un par un n'est évidemment pas une méthode efficace dans ce cas, puisque l'on pourrait utiliser la formule n^l. Mais au moins cela ne consomme quasiment aucune mémoire.

On considère maintenant les mots de Dyck, c'est-à-dire les mots en les lettres (et) bien parenthésés. La fonction ci-dessous engendre tous les mots de Dyck d'une longueur donnée, en utilisant la définition récursive disant qu'un mot de Dyck est soit vide, soit de la forme $(w_1)w_2$ avec w_1 et w_2 des mots de Dyck : :

```
sage: def dyck_words(l):
...       if l==0:
...           yield ''
...       else:
...           for k in range(l):
...               for w1 in dyck_words(k):
...                   for w2 in dyck_words(l-k-1):
...                       yield '('+w1+')'+w2
```

Voici tous les mots de Dyck de longueur 4 :

```
sage: list(dyck_words(4))
```

[()()()(), ()()(()), ()(())(), ()(()()), ()((())), (())()(), (())(()), (()())(), ((()))(), (()()()), (()(())),

On retrouve, en les comptant, une suite bien connue :

```
sage: [ sum(1 for w in dyck_words(l)) for l in range(10) ]
```

$$[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]$$

Exercice 47. Construire un itérateur sur l'ensemble $C(n)$ des arbres binaires complets à n feuilles (voir 12.1.2).

Indication : Sage n'a pas encore de structure de donnée native pour représenter des arbres binaires complets (cela viendra bientôt !). Pour l'instant, une manière simple de représenter des arbres est de définir une variable formelle `Leaf` pour les feuilles et une fonction formelle `Node` d'arité 2 (cela requiers une version 4.3 ou supérieure de Sage) :

```
sage: var("Leaf")               # variable formelle
```

Leaf

```
sage: function("Node", nargs=2) # fonction formelle d'arite 2
```

Node

Le deuxième arbre de la figure 12.1 peut alors être représenté par l'expression :

```
sage: tr = Node(Node(Leaf, Node(Leaf, Leaf)), Leaf)
```

12.3 Constructions

Nous allons voir maintenant comment construire de nouveaux ensembles à partir de briques de base. En fait, nous avons déjà commencé à le faire lors de la construction de $\mathcal{P}(\mathcal{P}(\mathcal{P}(\{1,2,3,4\})))$ dans la section précédente, ou pour construire des ensembles de cartes en § 12.1.

Considérons un produit cartésien un peu conséquent :

```
sage: C = CartesianProduct(Compositions(8), Permutations(20))
```

```
sage: C
```

```
Cartesian product of Compositions of 8, Standard permutations of 20
```

```
sage: C.cardinality()
```

31141145704660992000

Il ne serait évidemment pas envisageable de lister tous les éléments de ce produit cartésien.

Pour l'instant, la construction `CartesianProduct` *ignore les propriétés algébriques de ses arguments. Ce sera corrigé dans un avenir proche ; ainsi, le produit cartésien de deux groupes finis sera non seulement muni des opérations combinatoires usuelles, mais aussi de sa structure de groupe produit :*

```
sage: H = CartesianProduct([G,G])
```

```
sage: H.category()                    # todo: not implemented
```

Sets

Nous construisons maintenant la réunion de deux ensembles existants disjoints :

```
sage: C = DisjointUnionEnumeratedSets( [ Compositions(4), Permutations(3)] )
```

```
sage: C
```

```
Disjoint union of Family (Compositions of 4, Standard permutations of 3)
```

```
sage: C.cardinality()
```

14

```
sage: C.list()
```

[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4], [1, 2, 3], [1, 3, 2], [2, 1, 3], [

On peut faire une union de plus de deux ensembles disjoints, voire même d'une infinité d'entre eux. Nous allons construire l'ensemble de toutes les permutations, vu comme la réunion de tous les ensembles $P(n)$ des permutations de taille n. On commence par construire la famille infinie $F = (P(n))_{n\in\mathbb{N}}$:

```
sage: F = Family(NonNegativeIntegers(), Permutations)
```

```
sage: F
```

```
    Lazy family (Permutations(i))_i in Non negative integers
```

```
sage: F.keys()
```

```
                    Non negative integers
```

```
sage: F[1000]
```

```
                Standard permutations of 1000
```

On peut maintenant construire la réunion disjoint $\bigcup_{n\in\mathbb{N}} P(n)$:

```
sage: U = DisjointUnionEnumeratedSets(F)
```

```
sage: U
```

```
Disjoint union of Lazy family (Permutations(i))_i in Non negative integers
```

C'est un ensemble infini :

```
sage: U.cardinality()
```

$$+\infty$$

ce qui n'empêche pas d'itérer à travers ses éléments, quoiqu'il faille bien entendu interrompre le calcul à un moment donné :

```
sage: for p in U:
....:     print p
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

Note : on aurait pu construire l'ensemble ci-dessus directement avec

```
sage: U = Permutations()

sage: U
```

```
Standard permutations
```

12.3.1 Résumé

En résumé, Sage *fournit une bibliothèque d'ensembles énumérés usuels, qui peuvent être combinés entre eux par les constructions usuelles, ce qui donne une boîte à outil flexible. Il est de plus possible de rajouter en quelques lignes de code de nouvelles briques dans* Sage *(voir* `FiniteEnumeratedSets().example()`*). Cela est rendu possible par l'uniformité des interfaces et le fait que* Sage *soit basé sur un langage orienté objet. D'autre part, on peut manipuler des ensembles très grands, voire infinis, grâce aux stratégies d'évaluation paresseuse (itérateurs, ...).*

Il n'y a rien de magique : en arrière boutique, Sage *se contente d'appliquer les règles de calculs usuelles (par exemple, la cardinalité de $E \times E$ vaut $|E|^2$) ; la valeur ajoutée provient de la possibilité de manipuler des constructions compliquées. La situation est à rapprocher de l'analyse où, pour différencier une formule,* Sage *se contente d'appliquer les règles usuelles de différentiation des fonctions usuelles et de leur compositions, et où la valeur ajoutée viens de la possibilité de manipuler des formules compliquées. En ce sens,* Sage *implante un* calculus *sur les ensembles énumérés finis.*

12.4 Algorithmes génériques

Integer lists::

```
sage: IntegerVectors(10, 3, min_part = 2, max_part = 5, inner = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

sage: Compositions(5, max_part = 3, min_length = 2, max_length = 3).list()
[[1, 1, 3], [1, 2, 2], [1, 3, 1], [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2]]

sage: Partitions(5, max_slope = -1).list()
[[5], [4, 1], [3, 2]]

sage: IntegerListsLex(10, length=3, min_part = 2, max_part = 5, floor = [2, 4, 2]).l
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

sage: IntegerListsLex(5, min_part = 1, max_part = 3, min_length = 2, max_length = 3)
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1], [1, 2, 2], [1, 1, 3]]

sage: IntegerListsLex(5, min_part = 1, max_slope = -1).list()
[[5], [4, 1], [3, 2]]
```

```
    sage: c = Compositions(5)[1]
    sage: c
    [1, 1, 1, 2]

    sage: c = IntegerListsLex(5, min_part = 1)[1]

Species / decomposable classes
++++++++++++++++++++++++++++++

::

    sage: from sage.combinat.species.library import *
    sage: o   = var("o")

Fibonacci words::

    sage: Eps =  EmptySetSpecies()
    sage: Z0  =  SingletonSpecies()
    sage: Z1  =  Eps*SingletonSpecies()
    sage: FW  = CombinatorialSpecies()
    sage: FW.define(Eps + Z0*FW  +  Z1*Eps + Z1*Z0*FW)
    sage: FW

    sage: L = FW.isotype_generating_series().coefficients(15)
    sage: L

    sage: sloane_find(L)
    Searching Sloane's online database...
    [[45, 'Fibonacci numbers: F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1, F(2) = 1, ...', [0, 1, 1

    sage: FW3 = FW.isotypes([o]*4); FW3
    Isomorphism types for Combinatorial species with labels [o, o, o, o]

    sage: FW3.list()
    [o*(o*(o*(o*{}))), o*(o*(o*(({}*o)*{}))), o*(o*((({}*o)*o)*{})), o*((({}*o)*o)*(o*{})), o*((({

Binary trees::

    sage: BT = CombinatorialSpecies()
    sage: Leaf =  SingletonSpecies()
    sage: BT.define(Leaf+(BT*BT))
    sage: BT5 = BT.isotypes([o]*5)

    sage: BT5.list()
```

```
    [o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)), o*((o*(o*o))*o), o*(((o*o)*o)*o)

    sage: %hide
    sage: def pbt_to_coordinates(t):
    ...       e = {}
    ...       queue = [t]
    ...       while queue:
    ...           z = queue.pop()
    ...           if not isinstance(z[0], int):
    ...               e[z[1]._labels[0]-1] = z
    ...               queue.extend(z)
    ...       coord = [(len(e[i][0]._labels) * len(e[i][1]._labels))
    ...                       for i in range(len(e))]
    ...       return sage.geometry.polyhedra.Polytopes.project_1(coord)
    ...
    sage: K4 = Polyhedron(vertices=[pbt_to_coordinates(t) for t in BT.isotypes(range(5))
    sage: K4.show(fill=True).show(frame=False)
```

Juggling automaton::

```
    sage: F = SingletonSpecies()
    sage: state_labels = map(tuple, Permutations([0,0,1,1,1]).list())
    sage: states = dict((i, CombinatorialSpecies()) for i in state_labels)
    sage: def successors(state):
    ...       newstate = state[1:]+(0,)
    ...       if state[0] == 0:
    ...           return [(0, newstate)]
    ...       return [(i+1, newstate[0:i] + (1,) + newstate[i+1:])
    ...               for i in range(0, len(state)) if newstate[i] == 0]
    ...
    sage: for state in state_labels:
    ...       states[state].define(
    ...           sum( [states[target]*F
    ...                 for (height, target) in successors(state)], Eps ))
    ...
    sage: states
    {(1, 1, 0, 1, 0): Combinatorial species, (0, 1, 1, 0, 1): Combinatorial species, (1,

    sage: states[(1,1,1,0,0)].isotypes([o]*5).cardinality()
    165
```

Lattice points of polytopes
++++++++++++++++++++++++++

::

```
    sage: A=random_matrix(ZZ,3,6,x=7)
    sage: L=LatticePolytope(A)
    sage: L.plot3d()

    sage: L.npoints()  # should be cardinality!
    28

This example used PALP and J-mol


Words
=====

An infinite periodic word::

    sage: p = Word([0,1,1,0,1,0,1]) ^ Infinity
    sage: p
    word: 01101010110101011010101101010110101011 01...

The Fibonacci word::

    sage: f = words.FibonacciWord()
    sage: f
    word: 01001010010010100101001001010010010100 10...

The Thue-Morse word::

    sage: t = words.ThueMorseWord()
    sage: t
    word: 01101001100101101001011001101001100101 10...

A random word over the alphabet [0, 1] of length 1000::

    sage: r = words.RandomWord(1000,2)
    sage: r
    word: 00101010111011001100000001100001110111 00...

The fixed point of a morphism::

    sage: m = WordMorphism('a->acabb,b->bcacacbb,c->baba')
    sage: w = m.fixed_point('a')

    sage: w
    word: acabbbabaacabbbcacacbbbcacacbbbcacacbbac...
```

```
Their prefixes of length 1000::

    sage: pp = p[:1000]
    sage: ff = f[:1000]
    sage: tt = t[:1000]
    sage: ww = w[:1000]

A comparison of their complexity function::

    sage: A = list_plot([pp.number_of_factors(i) for i in range(100)], color='red')
    sage: B = list_plot([ff.number_of_factors(i) for i in range(100)], color='blue')
    sage: C = list_plot([tt.number_of_factors(i) for i in range(100)], color='green')
    sage: D = list_plot([r.number_of_factors(i) for i in range(100)], color='black')
    sage: E = list_plot([ww.number_of_factors(i) for i in range(100)], color='orange')
    sage: A + B + C + D + E


Construction of a permutation and builds its associated Rauzy diagram::

    sage: p = iet.Permutation('a b c d', 'd c b a')
    sage: p
    a b c d
    d c b a
    sage: r = p.rauzy_diagram()
    sage: r
    Rauzy diagram with 7 permutations
    sage: r.graph().plot()

Let us now construct a self-similar interval exchange transformation
associated to a loop in the Rauzy diagram::

    sage: g0 = r.path(p, 't', 't', 'b', 't')
    sage: g1 = r.path(p, 'b', 'b', 't', 'b')
    sage: g = g0 + g1
    sage: m = g.matrix()
    sage: v = m.eigenvectors_right()[-1][1][0]
    sage: T = iet.IntervalExchangeTransformation(p, v)

We can plot it and all its power::

    sage: T.plot()
    sage: (T*T).plot()
    sage: (T*T*T).plot()
```

```
Check the self similarity of T::

    sage: T.rauzy_diagram(iterations=8).normalize(T.length()) == T
    True

And get the symbolic coding of 0 using the substitution associated to the path::

    sage: s = g.orbit_substitution()
    sage: s.fixed_point('a')
    word: adbdadcdadbdbdadcdadbdadcdadccdadcdadbda...
```